```
#!/usr/bin/env python3
"""
Black Hole Orbiter (revised).

Improvements over original:
  1. Translated from MATLAB into Python, which opens the possibility of
     making a scientifically accurate game-engine with pygame visualizations
     while keeping low level control throughout for critical features.
  2. Simulates General Relativity Numerically, by referencing a 1975 paper
     "A Lorentz Covariant Treatment of the Kerr-Schild Geometry" by Gurses and Gursey.
     It is the only known way to fully simulate the General Theory of Relativity
     by using the only coordinate system that exactly linearizes the
     Einstein Field Equations, and does so without Supercomputing Clusters.
  3. Keeps the author's complete unifying vision of a Unified Field Theory intact.
     To do this requires a paradigm shift where every particle
     is also a rotating black hole, a field quantum, and an observer.
     This means one recovers Lorentz Covariant Motion in Special Relativity,
     Newtonian Force Laws of Universal Gravitation, Quantum Mechanics, and
     Multi-Particle Theory, with both mass and spin as particle invariants.

Author: Erik Jorgensen
Date: 2026-05-26 (revised), 2009-08-22 (original)
"""

import math
import numpy as np
from scipy.linalg import expm
import pygame
import sys

# ---------------------------------------------------------------------
# Minkowski metric  η_μν = diag(-1,+1,+1,+1)
# ---------------------------------------------------------------------

def minkowski_metric() -> np.ndarray:
    """Return η_μν with signature (−+++)."""
    return np.diag([-1.0, 1.0, 1.0, 1.0])


# ---------------------------------------------------------------------
# Change 4: Lorentz group via matrix exponential  (matches CGH.py)
# ---------------------------------------------------------------------

def lorentz_transform_matrix(Kx: float, Ky: float, Kz: float,
                             Jx: float, Jy: float, Jz: float) -> np.ndarray:
    """
    General Lorentz transformation  Λ = exp(M)  where M is the generator.

    K components are rapidities (boost parameters);
    J components are rotation angles.
    Convention identical to CGH.py:
        M = [[0,  Kx,  Ky,  Kz],
             [Kx,  0, -Jz,  Jy],
             [Ky,  Jz,  0, -Jx],
             [Kz, -Jy,  Jx,  0]]
    """
    M = np.array([[0.0,  Kx,   Ky,   Kz],
                  [Kx,  0.0,  -Jz,   Jy],
```

```
            [Ky,   Jz,  0.0,  -Jx],
            [Kz,  -Jy,   Jx,  0.0]])
    return expm(M)


def boost_matrix_from_3vel(v3) -> np.ndarray:
    """
    Pure Lorentz boost for 3-velocity v3 = (vx, vy, vz).

    Converts to rapidity  η = arctanh(|v|)  and feeds the unit-direction
    components into lorentz_transform_matrix as (Kx, Ky, Kz).
    No rotation parameters are used.
    """
    vx, vy, vz = float(v3[0]), float(v3[1]), float(v3[2])
    vmag = math.sqrt(vx*vx + vy*vy + vz*vz)
    if vmag < 1e-14:
        return np.eye(4)
    vmag_c = min(vmag, 1.0 - 1e-10)   # clamp away from light-speed
    rapidity = math.atanh(vmag_c)
    nx, ny, nz = vx/vmag, vy/vmag, vz/vmag
    return lorentz_transform_matrix(nx*rapidity, ny*rapidity, nz*rapidity,
                                    0.0, 0.0, 0.0)


def lorentz_inverse(Lambda: np.ndarray) -> np.ndarray:
    """
    Inverse of a Lorentz matrix:  Λ^{-1} = η Λ^T η.
    Works for any element of O(1,3).
    """
    eta = minkowski_metric()
    return eta @ Lambda.T @ eta


def rotation_to_align_spin(spin_unit) -> tuple:
    """
    Return (R, R_inv): 4×4 spatial rotation matrices built from
    lorentz_transform_matrix (zero boost params) that actively rotate
    *spin_unit* onto +Z.

    Derivation
    ----------
    Given unit vector n̂ we need R such that  R n̂ = ẑ.
    Rotation axis:  k̂ = (n̂ × ẑ) / |n̂ × ẑ|
    Rotation angle: θ  = arccos(n̂ · ẑ)
    Rotation vector sent to the generator: ω = θ k̂

    With ẑ = (0,0,1):
        n̂ × ẑ = (ny, −nx, 0)   →   ω = θ(ny, −nx, 0) / sinθ

    The Jx, Jy, Jz parameters of lorentz_transform_matrix encode the
    rotation vector directly (no boost, K=0), so the result is a pure
    spatial rotation that leaves the time component untouched.

    Special cases
    -------------
    n̂ ≈ +ẑ : identity (no rotation needed).
    n̂ ≈ −ẑ : π rotation around x̂.
    """
```

```
    nx, ny, nz = float(spin_unit[0]), float(spin_unit[1]), float(spin_unit[2])
    cos_theta = max(-1.0, min(1.0, nz))   # n̂ · ẑ = nz

    if cos_theta > 1.0 - 1e-12:           # already aligned with +Z
        return np.eye(4), np.eye(4)

    if cos_theta < -1.0 + 1e-12:          # anti-aligned: rotate π around x̂
        R     = lorentz_transform_matrix(0, 0, 0, math.pi, 0, 0)
        R_inv = lorentz_transform_matrix(0, 0, 0, -math.pi, 0, 0)
        return R, R_inv

    theta     = math.acos(cos_theta)
    sin_theta = math.sqrt(max(1.0 - cos_theta*cos_theta, 0.0))

    # ω = θ (ny, −nx, 0) / sinθ
    omx =  theta * ny / sin_theta
    omy = -theta * nx / sin_theta
    omz = 0.0

    R     = lorentz_transform_matrix(0, 0, 0,  omx,  omy, omz)
    R_inv = lorentz_transform_matrix(0, 0, 0, -omx, -omy, omz)
    return R, R_inv


# ---------------------------------------------------------------------
# 4-vector / 4-velocity helpers
# ---------------------------------------------------------------------

def interval(v: np.ndarray) -> float:
    """Spacetime interval  η_μν v^μ v^ν."""
    return -v[0]*v[0] + v[1]*v[1] + v[2]*v[2] + v[3]*v[3]


def normalize_4velocity(u: np.ndarray) -> np.ndarray:
    """Rescale so that  η_μν u^μ u^ν = -1."""
    n2 = interval(u)
    if abs(n2) < 1e-14:
        return u
    return u / math.sqrt(max(-n2, 1e-14))

def normalize_4spin(j: np.ndarray) -> np.ndarray:
    """Rescale so that  η_μν j^μ j^ν = 1."""
    n2 = interval(j)
    if abs(n2) < 1e-14:
        return j
    return j / math.sqrt(max(n2, 1e-14))

def four_velocity_from_3velocity(v3) -> np.ndarray:
    vx, vy, vz = float(v3[0]), float(v3[1]), float(v3[2])
    vmag = math.sqrt(vx*vx + vy*vy + vz*vz)
    vmag = min(vmag, 1.0 - 1e-10)
    gamma = 1.0 / math.sqrt(1.0 - vmag*vmag)
    return np.array([gamma, gamma*vx, gamma*vy, gamma*vz])


def three_velocity_from_4velocity(u: np.ndarray) -> np.ndarray:
    if abs(u[0]) < 1e-14:
        return np.zeros(3)
```

```
    return u[1:4] / u[0]


# ---------------------------------------------------------------------
# Change 5: Kerr metric from Lorentz-covariant tetrad, no active transform
#
# The tetrad (from CGH.py) with all Lorentz parameters = 0:
#
#   e[μ, a] = δ[μ, a]  +  (H/2) L^μ  l_a
#
# where  l_a = η_{av} L^ν  and  L^μ is the Kerr-Schild null vector.
# Spin axis is always Z in the source rest frame.
# This is a purely coordinate description — no passive boosts or rotations
# are applied, which would not correspond to any physical transformation.
# ---------------------------------------------------------------------

def _ks_r(a: float, x: float, y: float, z: float) -> float:
    """Kerr radial coordinate r solving  r^4 - (R²-a²)r² - a²z² = 0."""
    R2 = x*x + y*y + z*z
    Ra = R2 - a*a
    b3 = math.sqrt(Ra*Ra + 4.0*a*a*z*z)
    r2 = (Ra + b3) * 0.5
    return math.sqrt(max(r2, 0.0))


def _ks_H(m: float, a: float, r: float, z: float) -> float:
    """
    Kerr-Schild scalar  H = 2 m r³ / (r⁴ + a² z²).
    Factor of 2 is absorbed here (matches CGH.py convention) so that
    g_μν = η_μν + H L_μ L_ν  (rather than  η_μν + 2V λ_μ λ_ν).
    """
    denom = r**4 + a*a*z*z
    if denom < 1e-30:
        return 0.0
    return 2.0 * m * r**3 / denom


def _ks_L(a: float, r: float, x: float, y: float, z: float) -> np.ndarray:
    """
    Kerr-Schild null vector  L^μ  (contravariant), spin along +Z.
    Null with respect to both η and g.
    """
    L = np.zeros(4)
    denom = r*r + a*a
    L[0] = 1.0
    if denom > 1e-30:
        L[1] = (r*x + a*y) / denom
        L[2] = (r*y - a*x) / denom
    if r > 1e-30:
        L[3] = z / r
    return L


def kerr_metric_restframe(m: float, a: float, x3) -> np.ndarray:
    """
    Kerr metric g_μν at spatial position x3 = (x, y, z) in the source
    rest frame with spin axis along +Z.
```

```
    Built from the tetrad  e[μ,a] = I[μ,a] + (H/2) L^μ (η_{av} L^v)
    exactly as in CGH.py but with Lorentz parameters all zero (no active
    boost or rotation away from the standard Kerr-Schild chart):

        g[μ,v] = η[a,b] e[μ,a] e[v,b]
               = η[μ,v] + H L[μ] L[v]    (exploiting L·L = 0 w.r.t. η)

    The metric is stationary so only the spatial components of x3 matter.
    """
    x, y, z = float(x3[0]), float(x3[1]), float(x3[2])
    eta = minkowski_metric()
    r = _ks_r(a, x, y, z)
    H = _ks_H(m, a, r, z)
    L = _ks_L(a, r, x, y, z)          # L^μ  (upper)
    l = eta @ L                       # L_μ  (lower) = η_{μν} L^ν
    # Tetrad with Λ = I:  e[μ, a] = δ[μ,a] + (H/2) L[μ] l[a]
    e = np.eye(4) + 0.5 * H * np.outer(l, L)
    # g[μ,ν] = η[a,b] e[μ,a] e[ν,b]
    return np.einsum('ab,ua,vb->uv', eta, e, e)

def christoffel_fd(x3, m: float, a: float, eps: float = 1e-6) -> np.ndarray:
    """
    Christoffel symbols  Γ^μ_{αβ}  via centred finite differences.
    The metric is stationary, so only spatial derivatives are non-zero.
    Returns shape (4, 4, 4):  Gamma[mu, alpha, beta].
    """
    g0 = kerr_metric_restframe(m, a, x3)
    ginv = np.linalg.inv(g0)

    # ∂_i g_{μν}  for i ∈ {x, y, z}  →  dg[spatial_index, μ, ν]
    dg_spatial = np.zeros((3, 4, 4))
    for i in range(3):
        dx = np.zeros(3)
        dx[i] = eps
        gp = kerr_metric_restframe(m, a, np.array(x3) + dx)
        gm = kerr_metric_restframe(m, a, np.array(x3) - dx)
        dg_spatial[i] = (gp - gm) / (2.0 * eps)

    # Full  ∂_α g_{μν}:  α=0 (time) is zero for stationary metric
    dg = np.zeros((4, 4, 4))          # dg[α, μ, ν]
    dg[1] = dg_spatial[0]
    dg[2] = dg_spatial[1]
    dg[3] = dg_spatial[2]

    # Γ^μ_{αβ} = ½ g^{μν} (∂_α g_{νβ} + ∂_β g_{να} − ∂_ν g_{αβ})
    Gamma = np.zeros((4, 4, 4))
    for mu in range(4):
        for alpha in range(4):
            for beta in range(4):
                s = 0.0
                for nu in range(4):
                    s += ginv[mu, nu] * (dg[alpha, nu, beta]
                                         + dg[beta,  nu, alpha]
                                         - dg[nu, alpha, beta])
                Gamma[mu, alpha, beta] = 0.5 * s
    return Gamma
```

```
def geodesic_acceleration(u: np.ndarray, Gamma: np.ndarray) -> np.ndarray:
    """
    Geodesic 4-acceleration:  du^μ/dλ = −Γ^μ_{αβ} u^α u^β.
    """
    return -np.einsum('mab,a,b->m', Gamma, u, u)

def geodesic_spin_transport(j: np.ndarray, u: np.ndarray, Gamma: np.ndarray) -> np.ndarray:
    """
    Geodesic 4-spin parallel transport:  dj^μ/dλ = −Γ^μ_{αβ} j^α u^β.
    """
    return -np.einsum('mab,a,b->m', Gamma, j, u)

def Initialize_spin_vector(j: np.ndarray, u:np.ndarray):
    """
    Applies the physical constraint at t=0, such that
    j_μ u^μ = 0
    And it will evolve from there.
    """
    j[0] = (j[1]*u[1]+j[2]*u[2]+j[3]*u[3])/u[0]
    return j


# ---------------------------------------------------------------------
# Change 1 & 2: Particle class with worldline cache + proper-time state
# ---------------------------------------------------------------------

class Particle:
    """
    A gravitating body.  Integration is in proper time; coordinate time
    advances at rate  dt/dλ = u^0 (the Lorentz factor).

    The worldline cache stores snapshots at every integration step,
    ordered by (monotonically increasing) coordinate time.  Retarded
    positions are found by binary search on this cache.
    """

    def __init__(self, pos4: np.ndarray, vel4: np.ndarray, spin4: np.ndarray,
                 mass: float,
                 name: str = ""):
        """
        Parameters
        ----------
        pos4   : initial 4-position (t, x, y, z).
        vel4   : initial 4-velocity (will be normalised).
        spin4  : initial 4-spin     (will be normalised)
        mass   : gravitational mass (G = c = 1).
        spin_J : 3-vector whose *magnitude* is the Kerr spin parameter a
                 and whose *direction* is the spin axis in the lab frame.
                 Examples:
                   spin_J = (0, 0, 0.5)  -> a=0.5, spin along +Z
                   spin_J = (0.3, 0, 0)  -> a=0.3, spin along +X
        name   : optional label for diagnostics.
        """
        self.pos = pos4.astype(float).copy()
        self.vel = normalize_4velocity(vel4.astype(float).copy())
        self.spin = Initialize_spin_vector(spin4.astype(float).copy(),self.vel)
        self.mass = mass
        self.name = name
```

```
    self.proper_time: float = 0.0
    # Cache: list of dicts, sorted by ascending coordinate time.
    self._cache: list = [self._snapshot()]

# ------------------------------------------------------------------

def _snapshot(self) -> dict:
    return {
        't':   float(self.pos[0]),
        'pos': self.pos.copy(),
        'vel': self.vel.copy(),
        'spin': self.spin.copy(),
        'lam': self.proper_time,
    }

def record(self) -> None:
    """Append current state to cache."""
    self._cache.append(self._snapshot())

@property
def coord_time(self) -> float:
    return float(self.pos[0])

@property
def cache_t_min(self) -> float:
    return self._cache[0]['t']

@property
def cache_t_max(self) -> float:
    return self._cache[-1]['t']

# ------------------------------------------------------------------

def interpolate_at(self, t_query: float):
    """
    Linearly interpolate the cached worldline at coordinate time t_query.
    Returns (pos4, vel4).  Clamps to cache endpoints if out of range.
    """
    cache = self._cache
    if t_query <= cache[0]['t']:
        return cache[0]['pos'].copy(), cache[0]['vel'].copy(), cache[0]['spin'].copy(), cache[0]['lam']
    if t_query >= cache[-1]['t']:
        return cache[-1]['pos'].copy(), cache[-1]['vel'].copy(), cache[-1]['spin'].copy(), cache[-1]['lam']

    # Binary search for bracketing entries
    lo, hi = 0, len(cache) - 1
    while hi - lo > 1:
        mid = (lo + hi) >> 1
        if cache[mid]['t'] <= t_query:
            lo = mid
        else:
            hi = mid

    a0, a1 = cache[lo], cache[hi]
    dt = a1['t'] - a0['t']
    if dt < 1e-15:
        return a0['pos'].copy(), a0['vel'].copy(), a0['spin'].copy(), a0['lam']
```

```
    frac = (t_query - a0['t']) / dt
    pos = a0['pos'] + frac * (a1['pos'] - a0['pos'])
    vel = a0['vel'] + frac * (a1['vel'] - a0['vel'])
    spin = a0['spin'] + frac * (a1['spin'] - a0['spin'])
    lam =  a0['lam'] + frac * (a1['lam'] - a0['lam'])
    return pos, vel, spin, lam

# ---------------------------------------------------------------
# Change 1: retarded position via cache binary search
# ---------------------------------------------------------------

def retarded_position(self, observer_pos4: np.ndarray,
                      tol: float = 1e-10, max_iter: int = 64):
    """
    Find the retarded 4-position and 4-velocity of this particle as
    seen from observer_pos4 by binary-searching the worldline cache.

    Solves:  c·(t_obs − t_ret) = |x_src(t_ret) − x_obs|
             (with c = 1 in natural units)

    i.e. the separation 4-vector between source and observer is null.

    Parameters
    ----------
    observer_pos4 : 4-position of the observer at the *current* time.
    tol           : coordinate-time tolerance for the bisection.
    max_iter      : maximum bisection iterations.

    Returns
    -------
    pos_ret, vel_ret, spin_ret : 4-position, 4-velocity, 4-spin at the retarded event.
    """
    t_obs  = float(observer_pos4[0])
    x_obs  = observer_pos4[1:4]

    t_lo = self.cache_t_min
    t_hi = min(self.cache_t_max, t_obs)

    if t_hi <= t_lo:
        return self._cache[0]['pos'].copy(), self._cache[0]['vel'].copy(), self._cache[0]['spin'].copy(), self._cache[0]['lam']

    def residual(t_ret: float) -> float:
        """Positive when t_ret is too early, negative when too late."""
        pos_r, vel_r, spin_r, lam_r = self.interpolate_at(t_ret)
        dx = pos_r[1:4] - x_obs
        dist = math.sqrt(float(np.dot(dx, dx)))
        return (t_obs - t_ret) - dist

    f_lo = residual(t_lo)
    f_hi = residual(t_hi)

    # If the root is not bracketed, return the nearest endpoint
    if f_lo * f_hi > 0:
        t_ret = t_lo if abs(f_lo) < abs(f_hi) else t_hi
    else:
        for _ in range(max_iter):
            t_mid = 0.5 * (t_lo + t_hi)
            f_mid = residual(t_mid)
```

```
            if abs(f_mid) < tol:
                break
            if f_lo * f_mid <= 0.0:
                t_hi = t_mid
                f_hi = f_mid
            else:
                t_lo = t_mid
                f_lo = f_mid
        t_ret = 0.5 * (t_lo + t_hi)

    return self.interpolate_at(t_ret)


# ---------------------------------------------------------------------
# N-body acceleration using retarded cache + frame-boosted geodesics
# ---------------------------------------------------------------------

def compute_acceleration(particle: Particle,
                         sources: list) -> np.ndarray:
    """
    Total geodesic 4-acceleration of *particle* due to all *sources*.

    For each source the evaluation chain is:

      lab  --boost-->  source rest frame  --rotate-->  Z-aligned frame
           <--un-boost--                  <--un-rotate--

      1. Retarded (causal) position/velocity from cache.
      2. Boost lab -> source rest frame  (matrix-exponential, K params).
      3. Active spatial rotation aligning src.spin_unit with +Z
         (matrix-exponential, J params only, K=0).
         This is sandwiched between the boosts; it is NOT applied to the
         tetrad, which would be a passive relabelling, not a physical rotation.
      4. Tetrad Kerr metric evaluated with spin along Z  (untouched).
      5. Geodesic acceleration in the Z-aligned source frame.
      6. Inverse rotation: Z-aligned -> source rest frame.
      7. Inverse boost: source rest frame -> lab frame.
    """
    du_total = np.zeros(4)
    dj_total = np.zeros(4)

    for src in sources:
        # 1. Retarded source state ----------------------------------------
        x_ret, u_ret, j_ret, lam_ret = src.retarded_position(particle.pos)
        v3_src = three_velocity_from_4velocity(u_ret)

        # 2. Boost: lab -> source rest frame ------------------------------
        Lambda_to_src   = boost_matrix_from_3vel(-v3_src)
        Lambda_from_src = lorentz_inverse(Lambda_to_src)

        u_src  = Lambda_to_src @ u_ret
        dx_src = Lambda_to_src @ (particle.pos - x_ret)
        j_src = Lambda_to_src @ j_ret

        # 3. Decompose spin_J into magnitude (spin parameter a) + unit direction
        norm_j_src = normalize_4spin(j_src)
        spin_a = math.sqrt(max(interval(j_src), 1e-14))
        if spin_a > 1e-14:
```

```
            spin_unit = (norm_j_src[1], norm_j_src[2], norm_j_src[3])
        else:
            spin_unit = np.array([0.0, 0.0, 1.0])


        # 4. Active rotation: source rest frame -> Z-aligned frame -----------
        # rotation_to_align_spin returns the matrix that takes spin_unit -> Z.
        # Both the position offset and the 4-velocity must be rotated so
        # that the Kerr geometry (whose symmetry axis is Z in the tetrad)
        # is evaluated in the correct orientation.

        R_to_z, R_from_z = rotation_to_align_spin(spin_unit)

        dx_rot = R_to_z @ dx_src
        u_rot  = R_to_z @ u_src
        j_rot  = R_to_z @ j_src
        x_rel  = dx_rot[1:4]          # spatial offset in Z-aligned frame

        # 5. Kerr geodesic in Z-aligned source frame (tetrad, untouched) -
        Gamma  = christoffel_fd(x_rel, src.mass, spin_a)
        du_rot = geodesic_acceleration(u_rot, Gamma)
        dj_rot = geodesic_spin_transport(j_rot,u_rot,Gamma)

        # 6. Un-rotate: Z-aligned -> source rest frame -----------------------
        du_src_frame = R_from_z @ du_rot
        dj_src_frame = R_from_z @ dj_rot

        # 7. Un-boost: source rest frame -> lab frame ------------------------
        du_total += Lambda_from_src @ du_src_frame
        dj_total += Lambda_from_src @ dj_src_frame

    return du_total, dj_total


def Quantum(particle: Particle, dlambda: float):
    M = particle.mass
    v4 = particle.vel
    v3 = three_velocity_from_4velocity(v4)
    dr = np.sqrt(dlambda/M)
    deltaX = np.zeros((4))
    deltaX[0] = 0
    deltaX[1] = np.random.normal() * dr
    deltaX[2] = np.random.normal() * dr
    deltaX[3] = np.random.normal() * dr
    Lambda_from_rest = boost_matrix_from_3vel(v3)
    dX4 = Lambda_from_rest @ deltaX
    return dX4



# ---------------------------------------------------------------------
# Change 2: proper-time integration step
# ---------------------------------------------------------------------

def euler_step(particle: Particle, dlambda: float,
               sources: list) -> None:
    """Advance particle by one proper-time step  dλ  (forward Euler)."""
    dX4 = Quantum(particle, dlambda)
```

```
    particle.pos += dX4
    du, dj = compute_acceleration(particle, sources)
    particle.vel = particle.vel + du * dlambda
    particle.spin = particle.spin + dj * dlambda
    particle.pos = particle.pos + particle.vel * dlambda
    particle.proper_time += dlambda
    particle.record()


# ---------------------------------------------------------------------
# Change 3: coordinate-time-synchronised N-body loop
#
# All particles are advanced in *their own proper times* until every
# particle's worldline cache extends to the same target coordinate time.
# This guarantees that the retarded-time binary search always has
# sufficient history for any particle pair, regardless of how fast each
# particle's coordinate time ticks relative to its proper time.
# ---------------------------------------------------------------------

def advance_to_coord_time(particle: Particle, t_target: float,
                          dlambda: float, sources: list,
                          integrator=euler_step) -> None:
    """
    Integrate *particle* in proper time until  coord_time >= t_target.

    The inner loop uses  dlambda  proper-time steps.  Because
        dt_coord / dλ = u^0 = γ ≥ 1
    particles moving rapidly will take *fewer* proper-time steps to
    reach the same coordinate-time target.  The cache accumulates all
    intermediate states, providing a dense worldline for retarded lookups.
    """
    while particle.coord_time < t_target:
        integrator(particle, dlambda, sources)


def run_nbody(particles: list,
              dlambda: float,
              n_coord_steps: int,
              coord_step_size: float,
              integrator=euler_step,
              print_every: int = 10) -> dict:
    """
    N-body simulation loop with coordinate-time synchronisation.

    Parameters
    ----------
    particles       : list of Particle objects (initial conditions set).
    dlambda         : proper-time step per integration substep.
    n_coord_steps   : number of coordinate-time windows to simulate.
    coord_step_size : size of each coordinate-time window.
    integrator      : euler_step or rk4_step.
    print_every     : print diagnostics every this many coord steps.

    Returns
    -------
    histories : dict  {particle.name: [pos4, pos4, ...]}  sampled once
                per coord step.
    """
```

```
    histories = {p.name: [] for p in particles}
    timeline = []
    t_start = min(p.coord_time for p in particles)

    for step in range(n_coord_steps):
        t_target = t_start + (step + 1) * coord_step_size

        # Change 3: advance each particle to the common coordinate-time
        # target, filling its cache with dense proper-time samples.
        # Each particle sees the OTHER particles' already-cached worldlines
        # for the retarded-field evaluation.
        for particle in particles:
            sources = [p for p in particles if p is not particle]
            advance_to_coord_time(particle, t_target, dlambda,
                                  sources, integrator)

        # Diagnostics
        if step % max(1, print_every) == 0:
            t_now = particles[0].coord_time

            def fmt(v):
                return np.round(v, 4).tolist()

            info = []
            for p in particles:
                info.append(
                    f"{p.name}: pos={fmt(p.pos)} vel={fmt(p.vel)} spin={fmt(p.spin)}"
                )
            print(
                f"Step {step + 1:4d}/{n_coord_steps}  "
                f"t_coord≈{t_now:.3f}  "
                + " | ".join(info)
            )

        for p in particles:
            histories[p.name].append(p.pos.copy())
            timeline.append(t_target)

    # Return particles WITH caches intact
    particles_dict = {p.name: p for p in particles}

    return {
        'histories': histories,
        'particles': particles_dict,
        'timeline': timeline,
    }


# ---------------------------------------------------------------------
# FIX: build_observer_view takes particles (with caches), not histories
# ---------------------------------------------------------------------

def build_observer_view(particles_dict: dict,
                        observer_name: str,
                        dlambda: float) -> list:
    """
    Build observer frames sampled at observer's proper-time intervals.
```

```
    Each frame corresponds to one observer proper-time step (dlambda).
    For frame i: observer_proper_time = i * dlambda

    Returns
    -------
    frames : list of dicts
        [
          {
            't_obs': float,
            'lam_obs': float,
            'observer_pos_lab': 4-array,
            'observer_vel_lab': 4-array,
            'observer_spin_lab': 4-array,
            'sources': [
              {
                'name': str,
                'x_obsZ': 4-array,   # retarded source in observer rest frame + Z-aligned
                'u_obsZ': 4-array,
                'x_rel_spatial': 3-array,  # spatial part only
              },
              ...
            ]
          },
          ...
        ]
    """
    observer = particles_dict[observer_name]
    frames = []



    # Sample at observer's proper-time grid
    n_frames = int(observer.proper_time / dlambda) + 1

    for frame_idx in range(n_frames):
        lam_obs = frame_idx * dlambda

        # Look up observer state at this proper time
        # Find cache entry closest to lam_obs
        best_idx = 0
        best_diff = abs(observer._cache[0]['lam'] - lam_obs)
        for idx, snap in enumerate(observer._cache):
            diff = abs(snap['lam'] - lam_obs)
            if diff < best_diff:
                best_diff = diff
                best_idx = idx

        snap = observer._cache[best_idx]
        t_obs = snap['t']
        obs_pos_lab = snap['pos'].copy()
        obs_vel_lab = snap['vel'].copy()
        obs_spin_lab = snap['spin'].copy()

        # Boost lab -> observer rest frame
        v3_obs = three_velocity_from_4velocity(obs_vel_lab)
        Lambda_to_obs = boost_matrix_from_3vel(-v3_obs)

        sources_view = []
```

```
    for src_name, src in particles_dict.items():
        if src_name == observer_name:
            continue

        # 1. Retarded source event in lab frame
        pos_ret_lab, vel_ret_lab, spin_ret_lab, lam_ret_lab = src.retarded_position(obs_pos_lab)
        M_ret_lab = src.mass

        # 2. Express retarded event relative to observer in lab frame
        dx_lab = pos_ret_lab - obs_pos_lab

        # 3. Boost to observer rest frame
        dx_obs = Lambda_to_obs @ dx_lab
        u_obs = Lambda_to_obs @ vel_ret_lab
        j_src_obs = Lambda_to_obs @ spin_ret_lab

        # 4. Decompose spin_J into magnitude (spin parameter a) + unit direction
        j_obs = Lambda_to_obs @ obs_spin_lab
        norm_j_obs = normalize_4spin(j_obs)
        spin_a = math.sqrt(max(interval(j_obs), 1e-14))
        if spin_a > 1e-14:
            spin_unit = (norm_j_obs[1], norm_j_obs[2], norm_j_obs[3])
        else:
            spin_unit = np.array([0.0, 0.0, 1.0])

        # 5. Compute rotation that sends obs_spin_lab -> +Z
        R_to_z, R_from_z = rotation_to_align_spin(spin_unit)

        # 6. Rotate so observer's spin axis is +Z
        dx_obsZ = R_to_z @ dx_obs
        u_obsZ = R_to_z @ u_obs
        j_obsZ = R_to_z @ j_src_obs

        # 7. Compute quantum phase:
        Psi = np.real(np.exp(-1j*lam_ret_lab*M_ret_lab))
        sources_view.append(
            {
                'name': src_name,
                'pos': dx_obsZ[1:4].copy(),
                'field': Psi
            }
        )

    frames.append(
        {
            'time': lam_obs,
            'sources': sources_view,
        }
    )

  return frames


# ---------------------------------------------------------------------
# Example: two equal-mass bodies in approximate circular motion
# ---------------------------------------------------------------------
```

```
def run_example():
    """
    Two equal-mass bodies in approximate circular motion.
    p1 has spin along +Z (classic case).
    p2 has spin tilted into the XZ-plane to exercise the rotation sandwich.
    """
    M  = 1.0
    r0 = 25
    v_c = math.sqrt(M / r0)

    p1 = Particle(
        pos4   = np.array([0.0, -r0, 0.0, 0.0]),
        vel4   = four_velocity_from_3velocity((0.0,  0.0,  v_c/2)),
        mass   = M,
        spin4 = np.array([0.0, 0.0, 0.0, 0.5]),   # a=0.5, spin along +Z
        name   = "p1",
    )
    p2 = Particle(
        pos4   = np.array([0.0, r0, 0.0, 0.0]),
        vel4   = four_velocity_from_3velocity((0.0,  0.0,  -v_c/2)),
        mass   = M,
        spin4 = np.array([0.0, 0.3, 0.0, -0.4]),   # a=0.5, spin tilted in XZ-plane
        name   = "p2",
    )
    p3 = Particle(
        pos4=np.array([0.0, r0*3, -r0*4, -r0*12]),
        vel4=four_velocity_from_3velocity((0.0, 0.0, 0.95)),
        mass=M,
        spin4=np.array([0.0, 0.4, -0.3, 0.0]),  # a=0.5, spin tilted in YX-plane
        name="p3",
    )

    histories = run_nbody(
        particles       = [p1, p2, p3],
        dlambda         = 1,
        n_coord_steps   = 10,
        coord_step_size = 100,
        integrator      = euler_step,
        print_every     = 1,
    )
    print("Simulation finished.")
    return histories

#---------------------------------------
# PYTHON TEST SUITE FOR THE KERR TETRAD
#---------------------------------------

# Dirac Gamma Matrices:
T0 = np.array([[0,0,0,-1],[0,0,1,0],[0,-1,0,0],[1,0,0,0]])
X0 = np.array([[1,0,0,0],[0,-1,0,0],[0,0,1,0],[0,0,0,-1]])
Y0 = np.array([[0,0,0,1],[0,0,-1,0],[0,-1,0,0],[1,0,0,0]])
Z0 = np.array([[0,-1,0,0],[-1,0,0,0],[0,0,0,-1],[0,0,-1,0]])

# as cartesian(t,x,y,z) basis vectors
gamma = np.zeros((4,4,4))
gamma[0,:,:] = T0
gamma[1,:,:] = X0
gamma[2,:,:] = Y0
```

```
gamma[3,:,:] = Z0

def LorentzTransform(Kx,Ky,Kz,Jx,Jy,Jz):
    # Define general Lorentz Transformation matrix
    M = np.array([[0,Kx,Ky,Kz],
                  [Kx,0,-Jz,Jy],
                  [Ky,Jz,0,-Jx],
                  [Kz,-Jy,Jx,0]])
    # Compute Matrix Exponential of Generating Matrix
    Lorentz = expm(M)
    return Lorentz

def MinkowskiMetric():
    # The spacetime invariant interval
    N = np.zeros((4,4))
    N[0,0] = -1
    N[1,1] = 1
    N[2,2] = 1
    N[3,3] = 1
    return N

def radius(a,x,y,z):
    R2 = x*x + y*y + z*z
    Ra = R2 - a*a
    AZ = a*a*z*z
    b3 = np.sqrt(Ra*Ra + 4*AZ)
    r2 = (Ra + b3)/2
    r = np.sqrt(r2)
    return r

def Potential(m,a,r,z):
    AZ = a * a * z * z
    H1 = 2 * m * r * r * r
    H2 = r * r * r * r + AZ
    H = H1 / H2
    return H

def Vector(a,r,x,y,z):
    L = np.zeros(4)
    L[0] = 1
    L[1] = (r * x + a * y) / (r * r + a * a)
    L[2] = (r * y - a * x) / (r * r + a * a)
    L[3] = z / r
    return L
def Tetrad(m, a, t, x, y, z, Kx, Ky, Kz, Jx, Jy, Jz):
    X0 = np.array([t, x, y, z])
    Λ = LorentzTransform(Kx, Ky, Kz, Jx, Jy, Jz)  # Λ^μ{}_a
    X = np.linalg.solve(Λ, X0)                    # back to Kerr–Schild Cartesian

    eta = MinkowskiMetric()
    x0, y0, z0 = X[1], X[2], X[3]

    r = radius(a, x0, y0, z0)
    H = Potential(m, a, r, z0)
    L = Vector(a, r, x0, y0, z0)                  # L^μ

    # lower index: l_μ = η_{μν} L^ν
    l = eta @ L
```

```
    # transform to Lorentz frame index: L0_a = Λ^μ{}_a l_μ = (Λ.T @ l)_a
    L0 = Λ.T @ l

    # tetrad: e_μ{}^a = Λ_μ{}^a + (H/2) L_μ L0^a
    e0 = Λ.copy()
    e0 += 0.5 * H * np.outer(L, L0)

    return e0  # shape (4,4): e[μ,a]

def KerrMetric(m, a, t, x, y, z, Kx, Ky, Kz, Jx, Jy, Jz):
    e0 = Tetrad(m, a, t, x, y, z, Kx, Ky, Kz, Jx, Jy, Jz)  # e[μ,a]
    eta = MinkowskiMetric()                                # η[a,b]
    g = np.einsum('ab,ua,vb->uv', eta, e0, e0)
    return g


diff0 = 1e-9

def Connection(m, a, t, x, y, z, Kx, Ky, Kz, Jx, Jy, Jz):
    d = diff0
    dt = dx = dy = dz = d

    # metric at the point
    G = KerrMetric(m, a, t, x, y, z, Kx, Ky, Kz, Jx, Jy, Jz)
    g = np.linalg.inv(G)

    # metrics at shifted points
    Gt  = KerrMetric(m, a, t+dt, x,    y,    z,    Kx, Ky, Kz, Jx, Jy, Jz)
    Gt0 = KerrMetric(m, a, t-dt, x,    y,    z,    Kx, Ky, Kz, Jx, Jy, Jz)
    Gx  = KerrMetric(m, a, t,    x+dx, y,    z,    Kx, Ky, Kz, Jx, Jy, Jz)
    Gx0 = KerrMetric(m, a, t,    x-dx, y,    z,    Kx, Ky, Kz, Jx, Jy, Jz)
    Gy  = KerrMetric(m, a, t,    x,    y+dy, z,    Kx, Ky, Kz, Jx, Jy, Jz)
    Gy0 = KerrMetric(m, a, t,    x,    y-dy, z,    Kx, Ky, Kz, Jx, Jy, Jz)
    Gz  = KerrMetric(m, a, t,    x,    y,    z+dz, Kx, Ky, Kz, Jx, Jy, Jz)
    Gz0 = KerrMetric(m, a, t,    x,    y,    z-dz, Kx, Ky, Kz, Jx, Jy, Jz)

    # dG[μ,ν,ρ] = ∂_ρ g_{μν}
    dG = np.zeros((4, 4, 4))
    dG[:, :, 0] = (Gt - Gt0) / (2 * dt)
    dG[:, :, 1] = (Gx - Gx0) / (2 * dx)
    dG[:, :, 2] = (Gy - Gy0) / (2 * dy)
    dG[:, :, 3] = (Gz - Gz0) / (2 * dz)

    # B[μ,ν,σ] = ∂_μ g_{σν} + ∂_ν g_{σμ} - ∂_σ g_{μν}
    B = np.zeros((4, 4, 4))
    for mu in range(4):
        for nu in range(4):
            for sigma in range(4):
                B[mu, nu, sigma] = (
                    dG[sigma, nu, mu] +
                    dG[sigma, mu, nu] -
                    dG[mu, nu, sigma]
                )

    # Γ^ρ_{μν} = 1/2 g^{ρσ} B[μ,ν,σ]
    Gamma = 0.5 * np.einsum('rs,mns->rmn', g, B)
```

```
    return Gamma  # shape (4,4,4): Γ[ρ,μ,ν]

def CurvedGammas(m,a,t,x,y,z,Kx,Ky,Kz,Jx,Jy,Jz):
    e = Tetrad(m,a,t,x,y,z,Kx,Ky,Kz,Jx,Jy,Jz)   # e[μ,a]
    # Gamma[μ,i,j] = e[μ,a] * gamma[a,i,j]
    Gamma = np.einsum('ma,aij->mij', e, gamma)
    return Gamma

def Mass_Term():
    Mass_gamma = np.eye(4)
    for dim0 in range(0,4):
        Mass_gamma = Mass_gamma @ gamma[dim0,:,:] # Majorana Cartesian Gammas
    return Mass_gamma



# Tests
def MinkowskiMetricTest(m,a,t,x,y,z,Kx,Ky,Kz,Jx,Jy,Jz):
    e0 = np.linalg.inv(Tetrad(m,a,t,x,y,z,Kx,Ky,Kz,Jx,Jy,Jz))
    g = KerrMetric(m,a,t,x,y,z,Kx,Ky,Kz,Jx,Jy,Jz)
    h = np.zeros((4,4))
    for u in range(0,4):
        for v in range(0,4):
            for a0 in range(0,4):
                for b in range(0,4):
                    h[u,v] = h[u,v] + g[a0,b]*e0[u,a0]*e0[v,b]
    eta0 = MinkowskiMetric()
    print(h-eta0)
def KerrMetricTest(m,a,t,x,y,z,Kx,Ky,Kz,Jx,Jy,Jz):
    X0 = np.array([t, x, y, z])
    delta0 = LorentzTransform(Kx, Ky, Kz, Jx, Jy, Jz)
    X = np.linalg.inv(delta0) @ X0
    x0 = X[1]
    y0 = X[2]
    z0 = X[3]
    N = MinkowskiMetric()
    r = radius(a,x0,y0,z0)
    H = Potential(m,a,r,z0)
    L = Vector(a,r,x0,y0,z0)
    # Construct Kerr Spacetime
    G = np.zeros((4,4))
    for u in range(0,4):
        for v in range(0,4):
            G[u,v] = N[u,v] + H*L[v]*L[u]
    G0 = KerrMetric(m,a,t,x,y,z,Kx,Ky,Kz,Jx,Jy,Jz)
    print(G-G0)
def GammaFrameTest():
    eta = MinkowskiMetric()
    I = np.eye(4)
    for mu in range(0,4):
        for nu in range(0,4):
            Amat = gamma[mu,:,:]
            Bmat = gamma[nu,:,:]
            Cmat = np.matmul(Amat,Bmat)
            Cmat0 = np.matmul(Bmat,Amat)
            mat = (Cmat + Cmat0)/2
            Imat0 = I * eta[mu,nu]
            print(mat-Imat0)
```

```
#GammaFrameTest()
def VolumeTest(m,a,t,x,y,z,Kx,Ky,Kz,Jx,Jy,Jz):
    e0 = Tetrad(m, a, t, x, y, z, Kx, Ky, Kz, Jx, Jy, Jz)
    G = KerrMetric(m, a, t, x, y, z, Kx, Ky, Kz, Jx, Jy, Jz)
    print(np.linalg.det(e0)-1)
    print(np.linalg.det(G)+1)
    print(np.linalg.det(T0)-1)
    print(np.linalg.det(X0)-1)
    print(np.linalg.det(Y0)-1)
    print(np.linalg.det(Z0)-1)
def LorentzTetradTest(m,a,t,x,y,z,Kx,Ky,Kz,Jx,Jy,Jz):
    G0 = (Tetrad(m,a,t,x,y,z,0,0,0,0,0,0))
    c0 = (LorentzTransform(Kx,Ky,Kz,Jx,Jy,Jz))
    X = c0 @ np.array([t,x,y,z])#@ np.linalg.inv(c0)
    cG0 = Tetrad(m,a,X[0],X[1],X[2],X[3],Kx,Ky,Kz,Jx,Jy,Jz)
    cG1 = G0 @ c0
    print(cG0-cG1)

def TestAll():
    m = np.random.rand()
    a = np.random.rand()
    t = np.random.rand()
    x = np.random.rand()
    y = np.random.rand()
    z = np.random.rand()
    Kx = np.random.rand()
    Ky = np.random.rand()
    Kz = np.random.rand()
    Jx = np.random.rand()
    Jy = np.random.rand()
    Jz = np.random.rand()
    MinkowskiMetricTest(m, a, t, x, y, z, Kx, Ky, Kz, Jx, Jy, Jz)
    LorentzTetradTest(m, a, t, x, y, z, Kx, Ky, Kz, Jx, Jy, Jz)
    KerrMetricTest(m, a, t, x, y, z, Kx, Ky, Kz, Jx, Jy, Jz)
    VolumeTest(m, a, t, x, y, z, Kx, Ky, Kz, Jx, Jy, Jz)
    GammaFrameTest()

def RunAll():
    # Test proof of concept demo:
    TestAll()
    # Current simulation data:
    data = run_example()
    frames1 = build_observer_view(data['particles'], 'p1', 1)
    frames2 = build_observer_view(data['particles'], 'p2', 1)
    frames3 = build_observer_view(data['particles'], 'p3', 1)
    # Render Frame data from simulation:
    print('Building Scene from the Perspective of: p1')
    for frame in frames1:
        print(frame['time'], frame['sources'])
    print('Building Scene from the Perspective of: p2')
    for frame in frames2:
        print(frame['time'], frame['sources'])
    print('Building Scene from the Perspective of: p3')
    for frame in frames3:
        print(frame['time'], frame['sources'])

if __name__ == "__main__":
    RunAll()
```